Rolling Horizon Evolutionary Algorithms for General Video Game Playing

Raluca D. Gaina, Sam Devlin, Simon M. Lucas, Diego Perez-Liebana

***Abstract*—Game-playing Evolutionary Algorithms, specifically Rolling Horizon Evolutionary Algorithms (RHEA), have recently managed to beat the state of the art in win rate across many video games. However, the best results in a game are highly dependent on the specific configuration of modifications introduced over several papers, each adding additional parameters to the core algorithm. Further, the best previously published parameters have been found from only a few human-picked combinations, as the possibility space has grown beyond exhaustive search. This paper presents the state of the art in RHEA, combining all modifications described in literature, as well as new ones. We then use a parameter optimiser, the N-Tuple Bandit Evolutionary Algorithm, to find the best combination of parameters in 20 games from the General Video Game AI Framework. Further, we analyse the algorithm's parameters and some interesting combinations revealed through the optimisation process. Lastly, we find new state of the art solutions on several games by automatically exploring the large parameter space of RHEA.**

***Index Terms*—Evolutionary computation, rolling horizon, computational intelligence, artificial intelligence, games, general video game playing, real-time games**

I. Introduction

In this paper we revisit the application of game-playing Evolutionary Algorithms with a deeper analysis of algorithm modifications. We argue that automatic exploration (or algorithmic optimisation) of algorithm variations is essential for problems with large search spaces, although not exhaustive due to computation speed limitations. This optimisation process can further lead to insights into the algorithm being optimised, and as such we additionally conduct an in-depth analysis of the parameter space, while highlighting performance gain in various games.

There have been several recent advances in game-playing Evolutionary Algorithms [1], [2] and a multitude of modifications proposed to improve performance across a large number of games. The result is that the possibility space for algorithm configurations has grown beyond efficient manual optimisation. Although performing grid-search is sometimes possible for finding good values for some parameters [3], more recent works find the need to reduce more and more the number of parameter combinations chosen for analysis [4]. Therefore, the interesting insights into which variation of the algorithm is actually best are limited to human exploration of very small sections of the entirety of the search space.

Raluca D. Gaina, Simon M. Lucas, Diego Pérez-Liébana (School of Electronic Engineering and Computer Science, 10 Godward Square, Queen Mary University of London, Mile End Rd, London E1 4FZ, UK; email: {r.d.gaina,simon.lucas,diego.perez}@qmul.ac.uk), Sam Devlin (Microsoft Research Cambridge; email: Sam.Devlin@microsoft.com)

The specific novel application of Evolutionary Algorithms as game-playing methods (referred to as Rolling Horizon Evolutionary Algorithms, or RHEA) was introduced for the first time in 2013 by Perez et al. [5]. In the context of playing games, RHEA evolves, at every game step, a sequence of actions to play in the game; the first action of the best sequence found is played at the end of the evolutionary process and a new sequence is evolved for the subsequent game step. This base algorithm has been extended in several works. Gaina et al. [3] performed an in-depth analysis of the algorithm's main parameters (population size and individual length), generally finding that the higher the parameter values (even reaching the extreme of Random Search), the better RHEA performs across several games; this work further highlights an increase in performance with the increase of the available budget and correspondingly higher parameter values. Different population initialisation methods were explored in [6]; this work was important in highlighting the benefit of using different options in different game types, as some games saw increased performance with greedy initialisation, while others preferred a statistical approach instead. Furthermore, Gaina et al. tested in [7] various modifications and combinations with other techniques, which further pinpointed not only the difference in performance of certain parameter configurations across the different games, but also that the RHEA parameter space was already being expanded beyond the possibility of exhaustively exploring all parameter combinations. Some of these enhancements were further tested by Santos et al. [8] in General Video Game AI (GVGAI) and by Tong et al. [9] in MuJoCo's physical control tasks, both with great success. Finally, a study on dynamically adjusting individual length based on the fitness landscape observed during evolution [4] shows that some parameters might be conflicting with each other and cause poor performance in some games and suggests a need for carefully constructed parameter search spaces.

The work in this paper is carried out within the domain of general video game playing, which focuses on finding general-purpose Artificial Intelligence players that are able to play any game, even those unseen previously. The concepts behind this could be further extended to general AI which is able to solve any given task (as opposed to any given game), as methods developed for games have been shown to be applicable to wider domains, such as chemistry [10]. Two large categories of players can be differentiated in this domain: planning and learning. The latter requires training for several episodes on a game before it can figure out how to play it, which is often an expensive process leading to narrow results: the agent trained for one game would be unlikely to be able to play another

without significant training on the new game. The former category, which RHEA belongs to, refers to methods which work online, during the game, to search for the appropriate solutions. These methods require an internal model of the games (referred to as a forward model, or FM) to be able to simulate possible futures and effects of their actions. Although planning methods are more generally applicable, they face the drawback of the lack of a FM in some games, as this is not always feasible. The problem of learning any game's model is an active research area [11] which would make our methods even more widely applicable, even in complex commercial games; however, in this work we apply our algorithms to games which do have a model available.

In this context, Monte Carlo Tree Search (MCTS) had for a long time represented the state of the art in general video game playing. However, RHEA has been shown to outperform MCTS in multiple games in some of its variations [7], while other combinations of modifications led to significantly worse results. As highlighted by Lucas et al. [12], there can be a large difference in performance for the same base algorithm when using different parameters, and optimisation is essential. Ashlock et al. [13] emphasise this in the context of general game playing, where one single method (or single parameter configuration, in our approach) is unlikely to achieve high performance across all possible tasks. Our specific problem is additionally highly noisy: most games are stochastic and the same sequence of actions in a game could lead to different outcomes; furthermore, the algorithm itself is stochastic and may produce different outputs given the same game state.

In this paper, we use the N-Tuple Bandit Evolutionary Algorithm (NTBEA) [14] for optimising RHEA parameters, an algorithm which has shown robust high performance in noisy optimisation problems, even when compared with alternatives. It features high sample efficiency, fast convergence and good scaling for large search spaces [12]. Simulations of AI players on a multitude of games can be very expensive, therefore sample efficiency is key, making NTBEA suitable for optimising RHEA parameters. The algorithm has been previously successfully employed in several noisy optimisation problems, such as tuning game parameters [15] as well as AI game-player parameters [16], [12], [17]. A highly adaptive system which can optimise its parameters and structure so as to achieve best performance in various games could easily feed into a generic life-long learning system such as that presented in [18].

A summary of our contributions is as follows:

1) We give an overview of the current state of the art in Rolling Horizon Evolutionary Algorithms within the context of general video game playing.
2) We perform an in-depth analysis and optimisation of the algorithm's parameters with respect to its performance across the various games tested.
3) We find new configurations which outperform the previous state of the art on a range of GVGAI games.

II. Background

This section describes the two key concepts employed in this paper, the Rolling Horizon Evolutionary Algorithm (RHEA) and the N-Tuple Bandit Evolutionary Algorithm (NTBEA), as well as introducing the framework and game set used for experiments.

A. Rolling Horizon Evolution

RHEA utilises Evolutionary Algorithms (EA) to evolve an in-game sequence of actions at every game tick, with restricted computation time per execution. This subsection will describe the baseline algorithm, often referred to as *vanilla*; modifications applied are detailed in Section III.

In this application of EAs for game-playing, the genotype is described as a vector of integers of length L (action sequence length, or horizon), where each integer a is in the range $[0, N)$, with N being the maximum number of actions in a given game. This translates to a phenotype as a sequence of actions played in the game starting from state S_0, or, in other words, the behaviour of the player. In our implementation, the EA considers all individuals, and genes in the individuals, as legal and feasible. In order to evaluate an individual in this context, RHEA uses the forward model (FM) of the game, an internal model of the world, to simulate through the actions, one at a time. The game state reached at the end is then evaluated with a heuristic function h and this value becomes the fitness of the individual: therefore, we are evolving action sequences which lead to the best game outcome, limited to the exploration range L. The function h is always kept to a generic form throughout the experiments; this aims to maximise the game score, while favouring wins and discouraging losses, see Equation 1 (s is the game state being evaluated and r is the *reward* obtained from the game (or in-game *score*), normalised in $(0, 1)$).

$$h(s) = \begin{cases} 1 & \text{, win} \\ 0 & \text{, lose} \\ r & \text{, otherwise} \end{cases} \quad (1)$$

Using this method to evaluate individuals, the vanilla algorithm follows a typical EA process. It begins by initialising a population of P individuals of length L at random and evaluates them. At every generation, while budget is still available, it promotes E individuals directly to the next generation through elitism. It then generates P offspring by repeatedly selecting parents through tournament selection, crosses them with uniform crossover to create a child, and mutates the child through uniform mutation before adding it to the pool of offspring. The best $P - E$ individuals from both parents and offspring pools are added to the next generation and the process repeats. Typically, a budget of 40ms per game tick is given to the algorithm for real-time decision-making.

B. N-Tuple Bandit Evolutionary Algorithm

NTBEA is a model-based optimiser based on an Evolutionary Algorithm. It begins by randomly initialising a solution o, or with a given solution (referred to as *seed*, and the process as *seeding* the algorithm). It then evaluates one solution at a time, with the fitness function determined by the specific application. Small random noise is added to each solution's fitness value to distinguish between equally good solutions.

TABLE I: Game set including feature analysis. The last 3 columns show clusters as depicted in previous works; games with the same value are denoted as part of the same cluster. As [19] do not include all of these games in their study, column [20] shows the game indexes between which the missing games are placed by Mark Nelson (lower-higher); [21] shows more recent work clustering all GVGAI games.

Idx	Game	Stoch.	Rewards	Win	Lose	Levels	NPCs	Res.	Actions	[19]	[20]	[21]
0	Dig Dug	x	D	Puzzle/Kill	Timeout	L/Dense	E		Move+Shoot	4		5
1	Lemmings		D	Exit/Puzzle	Death	L/Dense	N		Move+Shoot	4		5
2	Roguelike	x	D	Exit	Death	L/Dense	E	x	Move+Shoot	4		4
3	Chopper	x	D+Disq	Kill	No-kill	L/Dense	E	x	Move+Shoot		g4-g1	2
4	Crossfire	x	N	Exit	Death	M/Dense	E		Move	2		4
5	Chase		D	Kill	Death	M/Sparse	F+E		Move	2		4
6	Camel Race		N	Exit	Timeout	L/Sparse	E		Move	2		3
7	Escape		N	Exit/Puzzle	Death	M/Dense			Move	2		3
8	Hungry Birds		Disq	Exit	Timeout	M/Sparse		HP	Move		g7-g10	3
9	Bait		N	Puzzle/Exit	Timeout	S/Sparse		x	Move	4		4
10	Wait for Breakfast		N	Puzzle	Timeout	M/Dense	N		Move	2		3
11	Survive Zombies	x	D	Timeout	Death	M/Dense	F+E		Move	3		4
12	Modality		N	Puzzle	Timeout	S/Dense			Move	3		4
13	Missile Command		D+Disq	Kill	No-kill	M/Sparse	E		Move+Shoot	3		2
14	Plaque Attack		D	Kill	No-kill	L/Dense	E		Move+Shoot	3		2
15	Sea Quest	x	D+Disq	Timeout	Death	M/Dense	F+E	x	Move+Shoot	3		2
16	Infection	x	D	Kill	Timeout	M/Dense	F+E		Move+Shoot	1		1
17	Aliens	x	D	Kill	Death	M/Dense	E		LR+Shoot	1		1
18	Butterflies	x	D	Kill	Timeout	M/Dense	F		Move	1		2
19	Intersection	x	D+Disq	Timeout	Death	L/Dense	E	HP	Move		g18-g17	1

1) n-tuple Model Update: The internal n-tuple model is then updated. In this context, an n-tuple is a list of length n, containing non-repeating numbers from 0 to $L-1$ (where L is the solution length); these numbers map to indexes in the solution evaluated. Statistics on all possible n-tuples are registered in the update step, keeping track of the number of times each n-tuple was sampled, how many times the indexed genes in the solution evaluated were observed to have those specific values, and the fitness value of the solution evaluated. We use 1, 2 and L values for n.

For example, consider solution (2, 3, 0, 0, 1) with fitness value 5. The 2-tuple (0,3) looks at indexes 0 and 3 in the solution, which map to specific gene values (2, 0). In the model, the number of times the (0,3) tuple was sampled is increased by 1, the number of times it mapped to gene values (2,0) is increased by 1, and value 5 is added to the sum value of the tuple. Similarly, all other combinations of 1, 2 and L tuples are added to the model as well.

2) Neighbourhood Evaluation: In the next step, several neighbours ($x = 50$) of the solution are generated through uniform random mutation, with probability $1/L$. The fitness of each neighbour is estimated based on the n-tuple model: all n-tuples are extracted from a neighbour o (let m be the total number of n-tuples), and statistics for each n-tuple are looked up in the internal model. For each n-tuple, we can then calculate its UCB value: if T_j is the j^{th} n-tuple, then $Q(T_j)$ is the average value observed for the n-tuple, $N(T_j)$ is the number of times the n-tuple was sampled, and $N(T_j, o)$ is the number of times the n-tuple was sampled and indexed the same gene values as in solution o. The constant $k = 2$ sets the focus of the algorithm, whether more exploitative or more exploratory. To obtain the overall UCB value for solution o, we calculate the average of UCB values for all n-tuples and add small random noise (maximum $\epsilon = 0.5$) to randomly break ties (see Equation 2).

$$UCB_o = \frac{1}{m}\sum_{j=1}^{m}\left(Q(T_j) + k \times \sqrt{\frac{\ln N(T_j)}{N(T_j, o)}}\right) + noise \quad (2)$$

3) Repeat: Finally, we choose the neighbour with the highest UCB value to be the next solution evaluated (o') and the process repeats for a set number of iterations. The final solution recommended is the one with the highest $Q(T)$ value averaged over all n-tuples. We refer the reader to [16] for more details on the NTBEA algorithm.

C. Framework

We use NTBEA to tune RHEA parameters within the General Video Game AI (GVGAI) framework [22]. GVGAI is a framework widely used in research [23] which features a corpus of over 100 single-player games and 60 two-player games. These are fairly small games, each focusing on specific mechanics or skills the players should be able to demonstrate, including clones of classic arcade games such as Space Invaders, puzzle games like Sokoban, adventure games like Zelda or game-theory problems such as the Iterative Prisoners Dilemma. All games are real-time and require players to make decisions in only 40ms at every game tick, although not all games explicitly reward or require fast reactions; in fact, some of the best game-playing approaches add up the time in the beginning of the game to run Breadth-First Search in puzzle games in order to find an accurate solution [23]. However, given the large variety of games (many of which are stochastic and difficult to predict accurately), scoring systems and termination conditions, all unknown to the players, highly-adaptive general methods are needed to tackle the diverse challenges proposed.

GVGAI includes several different tracks which tackle different problems: single-player planning [24], two-player planning [25] and single-player learning tracks focus on finding

general game-playing AI agents which would be capable of planning (with internal models of the world) or learning across all the games in the framework. More recently, level generation [26] (creating levels for any game) and rule generation [27] (creating rules for any given level) challenges were introduced as well, to push the limits of general game AI.

For the purpose of the experiments described in this paper, we will focus on the single-player planning track, although the work could easily be expanded to include two-player games.

D. Game set

We select 20 single-player games out of the larger GVGAI corpus, as previously analysed in several works. First introduced in [3] and described in Table I, the game set used in this study is sampled based on GVGAI competition entries performance across large subsets, so as to include games of varying difficulty. Additionally, half of the games are deterministic and half are stochastic, introducing additional noise to the parameter optimisation problem explored here.

The game table includes additional information about each game. They showcase varying reward structures, such as games with *no* rewards (with the possibility of gaining points on win/lose conditions only), games with *dense* rewards (multiple interactions with the environment result in a score change) or games with *discontinuous* rewards (a longer sequence of actions is required to obtain the reward). Four different types of winning conditions are featured, in which the player has to kill certain game objects (*Kill*), reach an exit point (*Exit*), wait for a timer to run out (*Timeout*) or complete a certain more precise sequence of actions (*Puzzle*, such as move a box onto a specific point). Three types of losing conditions are included, which result in the player losing if they run out of time (*Timeout*), die (*Death*) or fail to kill specific game objects (*No-kill*).

Additionally, the 5 levels included with each game vary in *size* (Large - *L*, Medium - *M* or Small - *S*) and *density* of interactive tiles (that is, tiles which produce some sort of effect when the player interacts with it, such as blocking the player's path, moving or getting destroyed). Some games include Non-Player Character (NPCs) that might either help the player (*F*), hurt the player (*E*) or have no direct influence on the player's win/lose condition or score (*N*) through their behaviour. The player may need to collect resources or pay particular attention to their avatar's hitpoints (HP). Finally, games vary in the actions available to the players (*Move* includes movement in all 4 directions, up, down, left and right; *LR* includes only left and right movement; a special *Shoot* action might be available in some games, with different effects). If the action chosen by the player at any game step is illegal or cannot reasonably be played (e.g. walking into a wall), it is automatically treated as *"do nothing"* by the game engine. All symbols mentioned here refer strictly to the table notation.

When discussing parameter choices, we will refer to games as similar based on the features described in Table I, or the clustering identified from previous works.

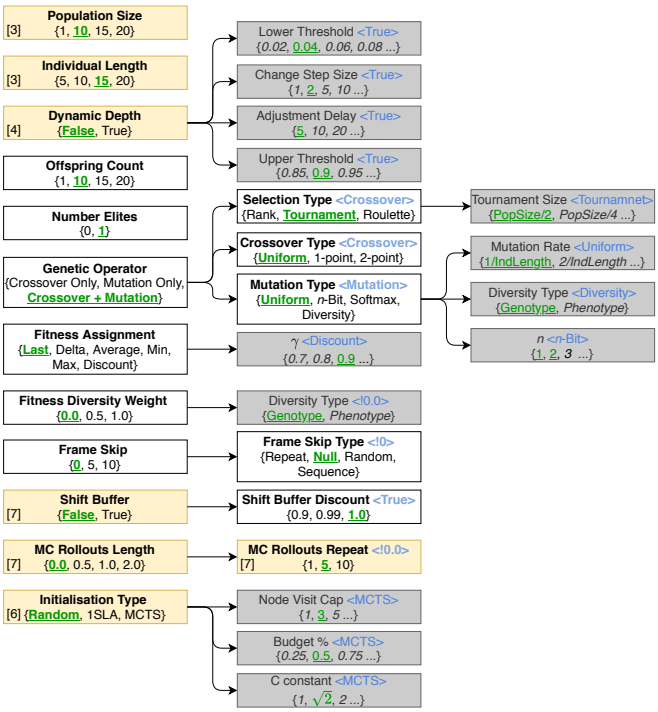

Fig. 1: Parameter search space, size 5.36×10^8 (excluding dark grey boxes). Possible values for a parameter in curly brackets, default value in underline green. Parameters not in the 1^{st} column are dependent on others (denoted with arrows from parent to dependent; parent value required for dependent to affect phenotype is noted in blue angled brackets). Dark-grey parameters are not included in the experiments for this paper (default values used instead). In yellow parameters previously analysed in literature, with citation.

III. RHEA Parameter Space

This section describes all the evolutionary algorithm hyper-parameters used for the experiments, some introduced in previous work [3], [6], [7], [4], as highlighted in Figure 1. Dependent parameters (2nd and 3rd column) are parameters that would not impact the phenotype without specific values taken by parent parameters, as detailed below.

A. Genetic operators

There are three main genetic operators used by the evolutionary algorithm in RHEA: crossover, selection and mutation. In our implementation, selection is only used to select parents for offspring, subsequent generations being formed directly with the best individuals from the current generation (with no further selection being applied). These three genetic operators each have several implementation options, as discussed below. A hyper-parameter controls which operators should be applied, with options of only using crossover (and selection), only using mutation, or using all three to first obtain an offspring from crossover and then mutate it as well.

Selection. Three types of selection are available in the system: tournament, roulette and rank. Tournament selection picks a percentage of the population ($t = 50\%$) randomly and then chooses the best individuals from these to reproduce.

Roulette selection chooses individuals with probabilities equal to their fitness. Rank selection first assigns inverse-ranks to all individuals in the population according to their fitness (the lowest fitness individual ranked first) and then chooses individuals with probabilities equal to their rank (reducing selection pressure). Selection is only used if crossover is enabled in the algorithm.

Crossover. Two types of crossover are available in the system: uniform and n-point. Uniform crossover selects genes from either of the parents with equal probability. n-point crossover randomly selects n points along the individuals which would split them in subsections, the offspring being then formed by alternatively choosing subsections of genes from the parents; we use 1 and 2 as possible values for n.

Mutation. Four types of mutation are available in the system: uniform, softmax, diversity and n-bits. Uniform mutation changes genes with equal probability ($m = 1/L$, where L is the individual length) and picks a different value uniformly at random. Softmax mutation uses the softmax equation (see Equation 3) to bias mutation towards the beginning of the individual, which causes the largest perturbation in the action sequence (changing any gene in the individual, in this context, also changes the meaning of all subsequent genes - therefore changes in the beginning of the genome have the largest impact in the phenotype). Diversity mutation keeps track of all values for all genes from all individuals explored during evolution and chooses to mutate the gene that has currently been explored the least, to the value for the gene that has been explored the least. Finally, n-bit mutation chooses n genes uniformly at random to mutate to a new and different random value; we use 1 and 2 as possible values for n.

$$\text{Softmax}(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \tag{3}$$

B. Fitness assignment

A key part of evaluating individuals, represented as action sequences, is the fitness assignment resulting from the phenotype interpretation (i.e. a sequence of game states the AI player traverses through the action sequence). If all game states traversed are evaluated with a heuristic function h (see Equation 1), then this array of values F corresponding to each of the game states can be translated to a fitness value in different ways: keeping only the value of the last game state reached: $F[L-1]$; keeping the difference between the value of the last state and the value of the first state, so state improvement value: $\Delta(F[L-1], F[0])$; keeping the average of all game state values: $\overline{F}$; keeping the minimum value, a pessimistic model: $min(F)$; keeping the maximum value, an optimistic model: $max(F)$; or keeping a discounted sum of all values: $\sum_{i=0}^{L} F[i] \times \gamma^i$, where $\gamma = 0.9$, which prioritises immediate rewards.

C. Initialisation

In the vanilla version, the algorithm is initialised with random individuals (all genes in all individuals are picked uniformly at random from all possible values). Different initialisation (or seeding) methods have been previously tested in conjunction with the vanilla algorithm with various success [6]. Both One Step Look Ahead (1SLA) and Monte Carlo Tree Search (MCTS) initialisation options, which have shown promise in various games in [6] are included in this system.

1SLA. This algorithm performs an exhaustive search of all possible actions in a given game state and picks the action which leads to the highest value for the following game state (evaluated with heuristic h and breaking ties randomly). To form an individual in the EA, this process is followed for each gene: the best action is chosen for the first gene, and the game state is advanced with the chosen action; this process is then repeated again for all subsequent genes until an action sequence of sufficient length is generated. If the end of the game is reached during the creation of an individual, the individual is padded with random actions until it meets the required length. For initialisation of a RHEA population, the first individual is created with the 1SLA algorithm and the rest become mutated from the first. Given the greedy approach, this reduces (and often completely removes) the randomness of the initial population, in order to begin search from a local optimum.

MCTS. This algorithm iteratively builds a search tree by selecting nodes in the tree to expand using the UCB1 formula, see Equation 4, where: constant $C = \sqrt{2}$, a is the chosen action from the set of possible actions $A(s)$, s is the current game state, $Q(s, a)$ is the value of choosing action a from state s, $N(s)$ is the number of times state s has been visited and $N(s, a)$ is the number of times state s has been visited and action a was chosen next. It then evaluates nodes with Monte Carlo simulations (a sequence of random actions up to a maximum tree depth L) starting from the newly expanded node and updates the statistics ($N(s)$, $N(s, a)$ and $Q(s, a)$ of all nodes traversed during an iteration with the value given by the heuristic h for the final game state reached after Monte Carlo simulations. This tree grows asymmetrically as MCTS balances between exploration of uncertain actions and exploitation of seemingly good actions. For initialisation of a RHEA population, MCTS is run for half of the entire thinking budget of the agent (leaving half available for evolution), and the first individual is selected by greedily traversing the tree created. As the tree would not be fully expanded, the path through the tree is capped when a node with less than 3 visits is reached and actions are added randomly up until individual length L; the rest of the individuals are mutated from the first.

$$a^* = \underset{a \in A(s)}{\arg\max} \left\{ Q(s, a) + C\sqrt{\frac{\ln N(s)}{N(s, a)}} \right\} \tag{4}$$

D. Frame skip

Frame skipping has become common practice in several Reinforcenemnt Learning works, and key in the success of specific applications [28], [29]: grouping N game states when making a decision, to increase the data available and reduce the frequency of decisions returned to only every N game states. Statistical forward planning approaches, on the other hand, usually make a new decision at every game tick, repeating their

search process in the very limited time. With this modification, we test if SFP methods can also benefit from a longer time for making decision by only returning an action every N game ticks, replying according to a specific strategy for the game ticks inbetween and using all the time inbetween decisions for planning the next move. We test 0 (no frame skip, decisions at every game tick), 5 and 10 as values for N and four different strategies for actions inbetween decisions: *repeat*, *null*, *random* and *sequence*. The *repeat* strategy simply repeats the previously decided action until a new action is decided. The *null* strategy plays `ACTION_NIL` (does nothing), which more closely mimics human player gameplay with pauses inbetween actions. The *random* strategy plays a random action and the *sequence* strategy continues playing the following actions in the best individual returned with the last decision.

A form of frame skip using the repeat strategy described above was previously tested in GVGAI by Perez et al. [30] with notable success in several games.

E. Shift buffer

This is a population management technique which avoids repeating the entire search process from scratch at every new game tick, which usually loses information gained in previous iterations of the algorithm; this is meant to make the algorithm more sample-efficient by retaining previous computation information. The shift buffer has been employed in several works and tested in GVGAI by Gaina et al. [7], and it works by keeping the final population evolved during one game tick to the next. However, as the first action of the best individual has just been played, all first actions from all individuals in the population are removed and a new random action is added at the end (a possible extension to this would keep only the individuals with the same first action as the one chosen). Additionally, there exists the option in our implementation to apply a discount to the fitness values of all individuals in the new population, which can be either 0.9, 0.99 or 1.0 (no discount applied); this would weaken the values of previously obtained sequences in the new context.

F. Dynamic depth

It is often the case that different games benefit from different algorithm parameters. In particular, the individual length has a high impact in the performance of the vanilla RHEA, as shown in [3]. This was mainly tied to the density of rewards in the various games in [4]: games with dense rewards generally benefit from shorter individuals which would allow for more generations and more statistics gathered to facilitate quick strategic reactions; as opposed to games with sparse or no rewards, where longer individuals are required in order to be able to find those rewards further ahead. This difference in reward density can also be observed at a more granular level, during the play-through of only one game: some areas of the game may contain more rewards, whereas others would require more exploration. Therefore, the dynamic depth modification presented in [4] is included in the system, which has the option to change the length of the individuals at every 5 game ticks: if the standard deviation of the fitness landscape observed previously falls below a threshold (0.04), decisions are considered to be uncertain without much variety in rewards observed and the individual length is increased by 2; if the opposite happens and the fitness landscape observed previously raises above a threshold (0.9), more generations are prioritised for more informed decision making in a highly varied environment and the individual length is decreased by 2 instead. All parameters for dynamic length adjustments were set based on [4] and could represent one point for further increasing the parameter search space in further studies.

G. MC rollouts

Lastly, we consider the further combination with MCTS, which has been very successful in many GVGAI games [20]. We have previously described MCTS initialisation, but concepts from MCTS can further be borrowed and integrated into RHEA, such as its Monte Carlo (MC) simulation phase. As described in [7], the evaluation process in RHEA may add MC rollouts of length $\{0.0$ (no rollouts used), 0.5, 1.0 or $2.0\}$ $\times$ L after advancing through the action sequence of length L represented by the individual; these may be repeated 1, 5 or 10 times for more statistics gathered. In order for this to be compatible with the fitness assignment modifications, the values of all game states traversed (or the average value for a particular game tick if there are repetitions performed) R are added at the end of the array of state values F obtained from the individual and all fitness assignment methods are applied to the combined array of values $(F + R)$ instead. The MC rollout repetition parameter is dependent on the rollout length: if the length is set to 0.0, then changing the number of rollout repetitions would have no effect on the phenotype.

IV. Experiments

Given the large number of parameter combinations, estimated at 5.36×10^8, it would take a significant amount of time to test each combination exhaustively in several games and with repetitions for statistical significance. Therefore we choose to analyse the different parameters indirectly through the evolutionary process described by an N-Tuple Bandit Evolutionary Algorithm (NTBEA). We ran NTBEA for 1500 iterations individually on each of the 20 games described in Section II to perform a search through the RHEA parameter space depicted in Figure 1, obtaining a (potentially different) parameter set recommendation for each game. Each individual evaluated by NTBEA would therefore be one parameter combination (18 individual length). We seed NTBEA with the previous state-of-the-art (SotA) parameter configuration for each game (see rows with citation in Table II). To evaluate each individual, we run RHEA with the specific parameter configuration on the given game, once in each of the 5 levels of the game and we use the average win rate on the 5 levels as individual fitness. To test the final configuration, we run it 100 times on the given game (20 times per level) and we additionally test the tuned parameter configuration on the entire set of 20 games, similarly with 100 runs per game.

TABLE II: RHEA best win rate (and standard error) recorded in all games. "opt" rows show NTBEA optimisation results, other rows show previously best recorded (with corresponding citation, highlighted in yellow). Parameters using default values (as per Figure 1) highlighted in green. Enhancements include values for dependants in brackets. Win-rates in bold are the higher values observed, if different.

Game	Win Rate	Parameters									Enhancements	
		Numerical				Nominal						
		P.Size	I.Len	Offspring	Elite	Init.	Selection	Crossover	Mutation	Fit.		
0	0% (0.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	0% (0.00)	1	20	15	1	MCTS	Rank	Uniform	2-bit	Last	SB(0.9); MC(0.5,1); Skip(Rep)	opt
1	**4%** (1.98)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	DD	[4]
	0% (0.00)	10	10	1	0	RND	Tourn.	1-point	2-bit	Last	SB(0.9); MC(0.5,5); Skip(Rep); DD	opt
2	0% (0.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	0% (0.00)	1	20	15	1	MCTS	Rank	Uniform	2-bit	Last	SB(0.9); MC(0.5,1); Skip(RND)	opt
3	100% (0.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	99% (0.99)	10	15	20	1	1SLA	-	-	Uniform	Disc.	SB(0.9); MC(1.0,5); DD	opt
4	10% (3.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	10% (3.00)	20	10	15	0	1SLA	-	-	2-bit	Disc.	SB(0.9); DD	opt
5	**13%** (3.39)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	3% (1.70)	10	20	20	1	RND	Tourn.	2-point	-	Average	MC(0.5,1)	opt
6	11% (3.13)	1	10	10	1	RND	Tourn.	Uniform	Uniform	Last	MC(0.5,10)	[7]
	41% (4.92)	15	15	1	1	MCTS	Tourn.	Uniform	Diversity	Disc.	SB(0.99); MC(1.0,5); Skip(Rep); Fit.Div(0.5)	opt
7	**46%** (4.98)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	MC(0.5,1)	[7]
	32% (4.665)	20	10	10	0	RND	-	-	Diversity	Disc.	SB(0.9); Fit.Div(0.5)	opt
8	**12%** (3.25)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	SB(0.9); MC(0.5,10)	[7]
	11% (3.13)	10	20	15	0	1SLA	Rank	2-point	2-bit	Max	SB(0.99); MC(2.0,5)	opt
9	**20%** (4.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	SB(0.9); MC(0.5,10)	[7]
	19% (3.923)	10	20	10	1	RND	Tourn.	1-point	Uniform	Max	SB(0.99); MC(1.0,5)	opt
10	78% (3.76)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	83% (3.76)	10	20	10	1	1SLA	Tourn.	1-point	-	Disc.	SB(0.99); MC(2.0,1); Skip(Null); DD	opt
11	55% (5.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	56% (4.97)	20	15	10	1	RND	Tourn.	Uniform	Uniform	Disc.	SB(0.99); MC(2.0,5); Skip(RND); DD	opt
12	**38%** (4.42)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	25% (4.33)	10	20	15	0	RND	Tourn.	1-point	Diversity	Max	MC(0.5,10); Skip(Seq); DD	opt
13	78% (4.18)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	86% (3.47)	15	20	15	1	RND	Tourn.	1-point	Softmax	Max	SB(0.9); MC(2.0,1)	opt
14	99% (1.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	100% (0.00)	15	20	1	1	1SLA	Rank	1-point	Diversity	Max	SB(1.0); MC(2.0,1); Skip(RND); DD	opt
15	65% (4.77)	1	5	1	1	RND	Tourn.	Uniform	Uniform	Last	SBer(0.9) MC(0.5,10)	[7]
	84% (3.66)	20	20	1	1	RND	Rank	Uniform	-	Average	SB(0.9); MC(2.0,1)	opt
16	**100%** (0.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	99% (0.99)	15	20	1	0	RND	-	-	2-bit	Last	SB(0.99); MC(0.5,10); Skip(RND); DD	opt
17	100% (0.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	100% (0.00)	15	20	20	0	RND	-	-	2-bit	Disc.	SB(0.9); DD	opt
18	**96%** (1.92)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	90% (3.00)	15	5	20	0	RND	Roulette	2-point	-	Disc.	SB(0.99); MC(2.0,5)	opt
19	100% (0.00)	10	15	10	1	RND	Tourn.	Uniform	Uniform	Last	-	[3]
	100% (0.00)	10	5	1	1	1SLA	Roulette	1-point	2-bit	Disc.	MC(2.0,5); DD	opt

All experiments were run on IBM System X iDataPlex dx360 M3 Server nodes, with one game per node, having one Intel Xeon E5645 processor core allocated to it and a maximum of 3GB of RAM of JVM Heap Memory. The runs took between 43 hours and 6 days to complete, including NTBEA tuning and final configuration testing; one run of a game can take up to 2000 game ticks to complete, with 1000 Forward Model calls per tick for AI decision making (plus game engine computations), the fastest game ending after 50 game ticks on average. The budget for all agents was set as 1000 Forward Model calls instead of time limits (which averages as the equivalent of 40ms in our tests), in order for the experiments to be consistent and replicable across different machines.

In the following sections we aim to analyse not only the performance of the optimised agents on the different games, but also the parameter space explored during the evolution and the parameter choices themselves. We hypothesise that similar games would lead to similar choices in parameters, which would differ across game types.

The paper presents and discusses the most interesting aspects observed, but all results, plotting scripts and additional

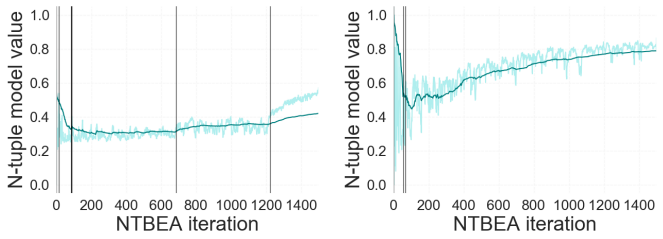

Fig. 2: Progression of solution fitness in the internal n-tuple model during NTBEA optimisation process in 2 games, *Missile Command* (left) and *Intersection* (right): plotting the value in the model for the solution evaluated at each iteration (light blue). The darker line indicates the value in the model for the seeded solution at each iteration (after n-tuple updates). Vertical lines indicate when the algorithm changes the best solution recommendation, based on its internal n-tuple model.

figures are available on Github[1]. We refer the reader to [3], [31] for a comparison of this algorithm with MCTS.

A. Optimisation Effectiveness

We first discuss the effectiveness of the optimisation. We summarise in Table II the results obtained on all 20 games used for tuning RHEA parameters with NTBEA. For each game, we present the parameter configuration of the previous state of the art (previous highest win rate recorded), its win rate and standard error; similarly, we present the optimised configuration for each game. Out of 20 games, 8 show worse results after optimisation, 6 observe similar results and in 6 we see an increased win rate. There are many games in which the win rate remains at or very close to 0%. This set of games (*Dig Dug*, *Lemmings*, *Roguelike*) remains too difficult for these methods to solve without more game-specific information or better exploration policies.

There are also several games which see win rates at, or very close to, 100% (*Intersection*, *Aliens*, *Infection*, *Chopper* and *Plaque Attack*). We do not see a decrease in performance in these games after optimisation (but a definite increase in *Plaque Attack* to 100% with several modifications in parameter choices, including using dynamic depth, 1SLA initialisation and random frame skip).

We do see several games improving performance: the win rate in *Sea Quest* increases from 65% to 84% by employing longer individual lengths, a larger population size, a shift buffer and MC rollouts. *Missile Command* sees an increase in win rate from 78% to 86% with a shift buffer, MC rollouts and a discounted fitness assignment. And performance in *Camel Race* increases from 11% to 41% by using *repetition* frame skip, a shift buffer and MC rollouts, amongst many configuration modifications. These 3 games do not immediately show common features as per Table I, with *Sea Quest* standing out due to its stochastic nature and dense environment, while the other two feature sparser deterministic environments.

We see a considerable decrease in performance in three of the games: *Butterflies* (from 96% to 90%), *Escape* (from 46% to 32%) and *Modality* (from 37.5% to 25%). As NTBEA was seeded with the previously best solution, we believe these are cases in which the noisy fitness evaluation was shown to be most harmful. Solutions are only evaluated once in each level of a game to obtain their true fitness value, while the final solutions are evaluated 20 times per level. Therefore, it is likely that the initial stochastic evaluation of the seed was misleading, leading the algorithm towards other areas of the search space. Additionally, since the value used in solution recommendations by NTBEA is based on its internal n-tuple model, this could also be a problem of credit assignment: all n-tuples are weighed equally, whereas they do not equally impact the phenotype. Future work will consider a better approach for tackling these difficult environments. A similar smaller decrease is also observed in *Lemmings* and *Bait* - all of these, except for *Butterflies*, are games with puzzle elements to them, which appear to be most difficult to optimise and estimate solution quality for, as they require more precise action sequences, with one move possibly making the game unsolvable, and therefore more precise evaluation. For these games, increasing the number of evaluations used in the fitness function would likely reduce the noise, as would changing the balance in the UCB calculations towards exploitation.

Finally, we highlight NTBEA's optimisation process progression in two games in Figure 2, *Missile Command* and *Intersection*. Both of these games see an upwards trend in solution quality, and they represent the games with the slowest and fastest convergence, respectively. We can observe that the algorithm settles on the solution for *Intersection* very quickly, before iteration 100, whereas it uses almost all computation budget for *Missile Command* to find the best option. This could be an indication of not only game difficulty, but also strategic depth: most parameter options work well and obtain very good performance in *Intersection*, while *Missile Command* poses a challenge at which not many options are successful and the finding of those few good configurations is more difficult. We note that the upward trend after settling on the best solution comes from the value of that best solution improving over time as more samples are added into the n-tuple model. Most games converged to a stable solution recommendation before the 1500 budget was exhausted. However, we expect solution quality to improve further with even more resources (e.g. Figure 2 shows Missile Command solution quality increasing and finding new bests up to very close to the given budget). Due to stochastic evaluations, different runs of the optimisation process may produce different results.

Overall, the game-specific optimised agents achieve win-rates of below 50% when tested on the entire set of games, which is not surprising in the general game playing context; the agents do not use any game-specific information. The best performing tuned agent is that for *Sea Quest* (53.4 average win rate on all 20 games), which shares different features with several other games; this appears to make the specific configuration more generally applicable than the others.

B. 1-tuple Analysis

Next, we look into the parameter space explored by NTBEA, starting with 1-tuples, or analysing each parameter and

[1]https://github.com/rdgain/ExperimentData/tree/NTBEA-RHEA-2019

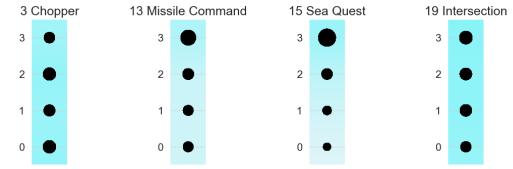

Fig. 3: 1-tuple: rollout length percentage parameter. Colors show average fitness for each data point, with blue being highest (1.0) and white being lowest (0.0). Each data point is highlighted with a black circle; the larger the circle, the more times that parameter value was sampled.

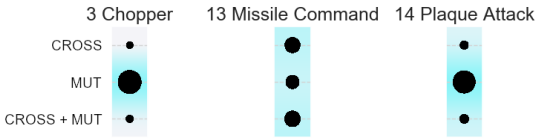

Fig. 4: 1-tuple: genetic operator parameter.

Fig. 5: 1-tuple: offspring count parameter.

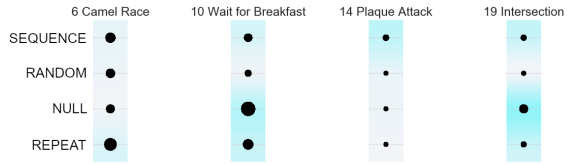

Fig. 6: 1-tuple: frame skip type parameter.

its preferred values in isolation in the different games tested; we use the term *prefer* to mean the value achieves highest win rate. We group together the solutions in which the parameter had the same value chosen and plot the average fitness of these solutions against parameter values. We note that the parameter may have not had a great influence in the fitness obtained, and we exclude the data points where the parameter had no influence at all, in the case of dependent parameters (see Figure 1). The resulting heatmaps show the fitness values observed for each parameter values, as well as how many times each parameter value was explored by NTBEA. The latter is given by the circle size in the figures presented; we cap the maximum number of occurrences of a data point at 100 and normalize all values in $[0, 1]$ for visualisation purposes.

17 games prefer the shift buffer turned on and to keep 1 elite between generations. Additionally, as previously seen in [3], most games prefer long individual length and large population sizes. 17 games further prefer the agent employing Monte Carlo rollouts at the end of its individual evaluation: *Chopper*, *Sea Quest* and *Missile Command* in particular prefer very long rollouts ($2.0 \times L$, see Figure 3); this could be due to these three games featuring different types of rewards and delays in obtaining rewards. The other similar game in terms of rewards, *Intersection*, does not show a particular preference in this parameter, achieving 1.0 fitness in all values.

In terms of genetic operators, most games prefer the agent to use both mutation and crossover in its evolutionary process. However, there are some exceptions: *Chopper* and *Plaque Attack* prefer to use mutation only, whereas *Missile Command* prefers options that do include crossover and more disturbance in its offspring (see Figure 4). Although these games are seen as similar in [21] and obtain high winning rates, the way the agents achieve their good performance does differ in these games, suggesting win-rate-based clustering methods could be improved by taking into account agent-based features.

When looking at the number of offspring (see Figure 5), *Survive Zombies*, *Missile Command* and *Chopper* prefer more. These games are quite similar in terms of features (win/lose conditions, level sizes, enemy NPCs) and are clustered together in [21]. However, in the same cluster, *Butterflies* and *Plaque Attack* don't show strong preference here - as opposed to the others, these two games have a smoother score progression, while *Missile Command* and *Chopper* show more delay in getting rewards, more actions are required from the player to find particular rewarding scenarios. *Sea Quest* is placed in a similar cluster by [19], but it shows opposite preference, for less offspring instead. In this game we see large discontinuous rewards as well as many smaller dense rewards - the larger variety in types of rewards could be what leads to favouring less solutions sampled to increase the number of generations in the evolutionary process, and to gain better insight into which reward type is preferable.

Lastly, we highlight that *Intersection* and *Wait for Breakfast* are the only games that benefit from *null* frame skipping (see Figure 6) - in both of these games it is essential to wait for specific events to happen (a way in the road to clear, or the waiter to arrive). *Plaque Attack* prefers *sequence* frame skipping, as plans evolved are precise enough in line with the constant stream of rewards. And *Camel Race* prefers *repeat* or *sequence*, with more frames skipped being better, which are effective strategies of exploring large sparse environments. Most other games dislike frame skipping and prefer a more fine-grained search; however, we note that the search space for this parameter is very coarse and it might be that more games could benefit from *some* or dynamic frame skipping.

C. 2-tuples Analysis

Similarly as with 1-tuples, we can look at how combinations of parameters affect overall solution fitness. In this section we group together solutions which had the same values for each parameter combination, while eliminating the data points where either one or both of the parameter values did not impact solution phenotype, in case of dependent parameters (see Figure 1). We plot each parameter against all others in

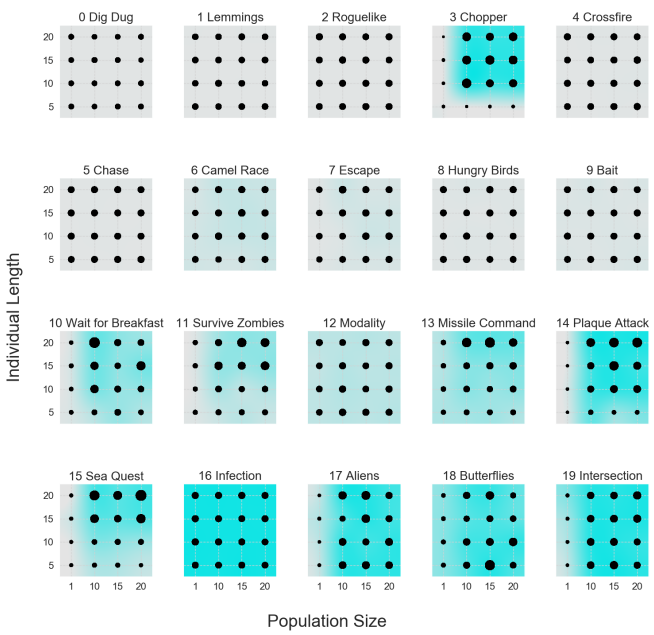

Fig. 7: 2-tuple: individual length and population size. Colors show average fitness for each data point, with blue being highest (1.0) and white being lowest (0.0). Each data point is highlighted with a black circle; the larger the circle, the more times that combination of values was sampled.

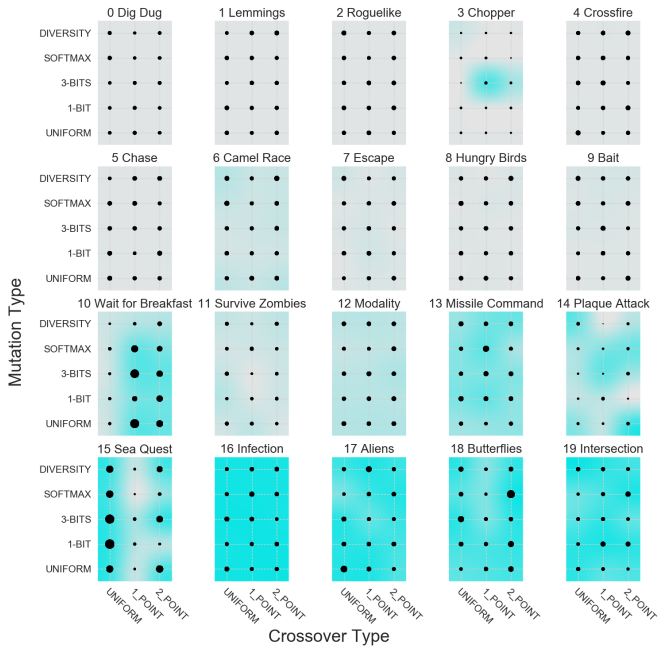

Fig. 8: 2-tuple: mutation type and crossover type.

the different games tested, each data point representing the average fitness observed in the respective group of solutions. We further add black circles on each data point to highlight the number of times each combination was explored by NTBEA during the optimisation process.

The first thing that stands out in all resulting figures is that NTBEA explores the best combinations the most, while mostly ignoring less promising options. This can be seen as a direct confirmation of the effectiveness of the bandit-based approach, but also as a potential point of improvement: due to the nature of the very noisy optimisation, it might be beneficial to obtain more accurate estimates of some data points which do not immediately stand out as the best: as discussed in a previous section, it was the case in several games that the optimised solution ended up performing worse than the initial solution given to the algorithm, which could have been avoided had a more accurate evaluation of solution quality been done.

In Figure 7 we can observe the combination of individual length L and population size P parameters. We've previously observed that longer individuals lead to higher fitness values, and similar for larger population sizes. It is interesting to see that this holds true also for the combination of L and P, although specific combinations achieve better results in some games (such as $L = 20$ and $P = 15$ in *Chopper*).

Another interesting parameter combination to discuss is that of mutation type and crossover type, shown in Figure 8, which largely decides how offspring are created at each generation. Although the overall fitness of solutions differs, games *Hungry Birds* and *Plaque Attack* show a similar distribution of good or bad quality combinations: in particular, 1-point crossover does not agree with diversity mutation, and 2-point crossover does not agree with bit-mutation. This could largely be due to the specific modifications n-point crossover wishes to generate, which are modified unexpectedly by bit-mutation. However, these two games singled out here do not appear to have much in common according to our feature descriptions and clustering in Table I; it is thus interesting to find game similarities beyond those given by traditionally-employed features.

V. Conclusion

In this paper we use the N-Tuple Bandit Evolutionary Algorithm (NTBEA) to optimise the performance of Rolling Horizon Evolutionary Algorithm (RHEA) in 20 GVGAI games, by modifying the configuration of RHEA's 18 parameters. The various values possible for all parameters form a large search space of 5.36×10^8, which makes manual optimisation or exhaustive search difficult with limited compute, thus we choose to use NTBEA to attempt to improve the win rate of the agent in each of the 20 games.

As a result of the optimisation, the performance increases in several games. However, puzzles appeared to be the games where NTBEA struggled to estimate the quality of different agent configurations and the solution returned was worse than the state of the art, although NTBEA's evolutionary process was run with SotA as the initial solution. The optimisation process differed in the games tested, NTBEA being able to converge in under 100 iterations in some games, while taking most of its 1500 iteration budget to find good solutions in others: this strengthens the idea that one specific method is unlikely to perform well across all games, and that games might require specialised parameter search spaces to ensure fast optimisation or even the possibility of a high-performing solution being found.

We further analysed RHEA's parameters through the evolutionary process, by looking at some 1-tuples and 2-tuples and the values explored for each. Several games with similar features in common were found to prefer similar parameter values, although exceptions do exist of game clusters shown in parameter values, but not in the traditional game features considered. This suggests that game clustering methods can be further enhanced by considering agent-based features.

To further expand on the work carried out here, we propose exploring larger and more complex search spaces, with an enhanced NTBEA which is able to handle tree-structures: we've seen several parameters dependent on others and optimisation would be more sample-efficient if this was taken into account during the evolutionary process. NTBEA parameters can be better adjusted as well: decreasing the exploration constant over time could lead to better results. More enhancements can also be added into the system, as well as optimising RHEA on a larger set of games (including multi-player games and games with hidden information), with the possibility of testing approaches at optimising a generally applicable player. Moreover, other optimisers could be tested, such as Bayesian Optimisation, to observe difference in results and insights gained, similar to the approach taken in [31]. Lastly, information gathered during optimisation and in-depth analysis can be used for designing hyper-parameter methods which would be able to identify game features, relate these to previously seen situations and adapt to new unknown environments.

ACKNOWLEDGMENTS

This work was funded by the EPSRC CDT in Intelligent Games and Game Intelligence (IGGI) EP/L015846/1

REFERENCES

[1] H. Baier and P. I. Cowling, "Evolutionary MCTS with Flexible Search Horizon," in *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[2] N. Justesen, T. Mahlmann, S. Risi, and J. Togelius, "Playing Multiaction Adversarial Games: Online Evolutionary Planning Versus Tree Search," *IEEE Transactions on Games*, vol. 10, no. 3, pp. 281–291, 2017.

[3] R. D. Gaina, J. Liu, S. M. Lucas, and D. P. Liébana, "Analysis of Vanilla Rolling Horizon Evolution Parameters in General Video Game Playing," in *Springer L.N. in Comp. Science, Applications of Evolutionary Computation, EvoApplications*, no. 10199, 2017, pp. 418–434.

[4] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "Tackling Sparse Rewards in Real-Time Games with Statistical Forward Planning Methods," in *AAAI Conference on Artificial Intelligence (AAAI-19)*, vol. 33, 2019, pp. 1691–1698.

[5] D. Perez-Liebana, S. Samothrakis, S. M. Lucas, and P. Rolfshagen, "Rolling Horizon Evolution versus Tree Search for Navigation in Single-Player Real-Time Games," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2013, pp. 351–358.

[6] R. D. Gaina, S. M. Lucas, and D. P. Liébana, "Population Seeding Techniques for Rolling Horizon Evolution in General Video Game Playing," in *Proceedings of the Congress on Evolutionary Computation*, June 2017, pp. 1956–1963.

[7] ——, "Rolling Horizon Evolution Enhancements in General Video Game Playing," in *Proceedings of IEEE Conference on Computational Intelligence and Games*, Aug 2017, pp. 88–95.

[8] B. Santos, H. Bernardino, and E. Hauck, "An Improved Rolling Horizon Evolution Algorithm with Shift Buffer for General Game Playing," in *2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)*. IEEE, 2018, pp. 31–316.

[9] X. Tong, W. Liu, and B. Li, "Enhancing Rolling Horizon Evolution with Policy and Value Networks," in *2019 IEEE Conference on Games (CoG)*. IEEE, 2019, pp. 1–8.

[10] M. H. Segler, M. Preuss, and M. P. Waller, "Planning Chemical Syntheses with Deep Neural Networks and Symbolic AI," *Nature*, vol. 555, no. 7698, p. 604, 2018.

[11] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *arXiv preprint arXiv:1911.08265*, 2019.

[12] S. M. Lucas *et al.*, "Efficient Evolutionary Methods for Game Agent Optimisation: Model-Based is Best," *arXiv preprint arXiv:1901.00723*, 2019.

[13] D. Ashlock, D. Perez-Liebana, and A. Saunders, "General Video Game Playing Escapes the No Free Lunch Theorem," in *2017 IEEE Conf. on Computational Intelligence and Games (CIG)*. IEEE, 2017, pp. 17–24.

[14] K. Kunanusont, R. D. Gaina, J. Liu, D. Perez-Liebana, and S. M. Lucas, "The N-Tuple Bandit Evolutionary Algorithm for Automatic Game Improvement," in *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2017, pp. 2201–2208.

[15] K. Kunanusont, S. M. Lucas, and D. Perez-Liebana, "Modelling Player Experience with the N-Tuple Bandit Evolutionary Algorithm," in *Artificial intelligence and Interactive Digital Entertainment (AIIDE)*, 2018.

[16] S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," in *2018 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2018, pp. 1–9.

[17] I. Bravi, S. Lucas, D. Perez-Liebana, and J. Liu, "Rinascimento: Optimising Statistical Forward Planning Agents for Playing Splendor," *arXiv preprint arXiv:1904.01883*, 2019.

[18] R. D. Gaina, S. M. Lucas, and D. Perez-Liebana, "Project Thyia: A Forever Gameplayer," in *IEEE Conference on Games (COG)*, 2019, pp. 1–8. [Online]. Available: https://ieeexplore.ieee.org/document/8848047

[19] P. Bontrager, A. Khalifa, A. Mendes, and J. Togelius, "Matching Games and Algorithms for General Video Game Playing," in *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2016, pp. 122–128.

[20] M. J. Nelson, "Investigating Vanilla MCTS Scaling on the GVG-AI Game Corpus," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2016, pp. 1–7.

[21] M. Stephenson, D. Anderson, A. Khalifa, J. Levine, J. Renz, J. Togelius, and C. Salge, "A Continuous Information Gain Measure to Find the Most Discriminatory Problems for AI Benchmarking," *arXiv preprint arXiv:1809.02904*, 2018.

[22] D. Perez-Liebana, S. M. Lucas, R. D. Gaina, J. Togelius, A. Khalifa, and J. Liu, *General Video Game Artificial Intelligence*. Morgan and Claypool Publishers, 2019. [Online]. Available: https://gaigresearch.github.io/gvgaibook/

[23] D. Perez-Liebana, J. Liu, A. Khalifa, R. D. Gaina, J. Togelius, and S. M. Lucas, "General Video Game AI: a Multi-Track Framework for Evaluating Agents Games and Content Generation Algorithms," *IEEE Transactions on Games*, vol. 11, no. 3, pp. 195–214, Sep 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8664126

[24] D. Perez-Liebana *et al.*, "General Video Game AI: Competition, Challenges and Opportunities," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[25] R. D. Gaina, A. Couëtoux, D. J. Soemers, M. H. Winands, T. Vodopivec, F. Kirchgessner, J. Liu, S. M. Lucas, and D. Perez-Liebana, "The 2016 Two-Player GVGAI Competition," *IEEE Transactions on Games*, vol. 10, no. 2, pp. 209–220, June 2018.

[26] A. Khalifa, D. Perez-Liebana, S. M. Lucas, and J. Togelius, "General Video Game Level Generation," in *Proc. of the Genetic and Evolutionary Computation Conference 2016*. ACM, 2016, pp. 253–259.

[27] A. Khalifa, M. C. Green, D. Perez-Liebana, and J. Togelius, "General Video Game Rule Generation," in *2017 IEEE Conference on Computational Intelligence and Games (CIG)*. IEEE, 2017, pp. 170–177.

[28] A. Braylan, M. Hollenbeck, E. Meyerson, and R. Miikkulainen, "Frame Skip is a Powerful Parameter for Learning to Play Atari," in *Workshops at the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[29] M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling, "Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents," *Journal of Artificial Intelligence Research*, vol. 61, pp. 523–562, 2018.

[30] D. P. Liébana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas, "Introducing Real World Physics and Macro-Actions to General Video Game AI," in *Proceedings of IEEE Conference on Computational Intelligence and Games*, Aug 2017, pp. 248–255.

[31] R. D. Gaina, C. F. Sironi, M. H. Winands, D. P. Liébana, and S. M. Lucas, "Self-Adaptive Rolling Horizon Evolutionary Algorithms for General Video Game Playing," in *IEEE Conference on Games (CoG)*, 2020.